数据结构期末汇报

实习 2.4 马踏棋盘，实习 6.4 平衡二叉树操作的演示

袁晨圃，李知谦，邱子陶

University of Chinese Academy of Sciences

2025-06-25

1. 马踏棋盘问题演示

1.1 算法

1.1.1 暴力搜索算法

暴力算法实现 solve_brute_force(Point start):

通过手动维护一个栈记录走过的路径结点坐标与正在尝试的方向，当无路可走时利用栈进行回溯从而尝试新的路径。

```
class SimpleStack {
private:
    T* base;
    int top;
    int capacity;  // 当前容量

    void expand() { ...
public:
    SimpleStack(int initial_capacity = 100) : base(nullptr), top(-1), capacity(initial_capacity)
{ ...                            // 构造
    ~SimpleStack() { ...            // 销毁
    void push(const T& value) { ... // 入栈
    void pop() { ...                // 出栈
    T& peek() { ...                 // 读栈顶
    bool empty() const { ...        // 判断是否栈空
    int size() const { ...          // 返回栈元素个数
```

```
struct Node {
    Point pos;   // 当前坐标
    int move_index;  // 当前正在尝试的方向
};
```

1.1 算法

为了可视化搜索的过程，我们决定记录过程中每一步试探（包括回溯）。

最开始算法中每步都存储完整的棋盘（二维数组），但每步都存储一个棋盘带来的内存占用过大。

后来我们决定只使用起始点与终止点的坐标对记录每一步的行动，通过 stepNext 标签记录该步是前进还是回溯。

```
struct Arrow {
    Point start, end;
    bool stepNext; // 前进为1，后退为0
};

void Print_board(Board);

using Path = std::vector<Arrow>; // 一条可行路径

// 算法返回值：
return std::vector<Path> //允许返回多条可行路径
```

但即便如此，暴力搜索算法的大量路径试探仍会带来无法承受的内存开销。

1.1 算法

1.1.2 贪心算法

贪心算法实现 `solve_heuristic(Point start)`:

基于 H. C. von Warnsdorf 于 1823 提出的 Warnsdorf's Rule ——每步选择可移动方向最少得位置移动。该算法可以以极快的速度给出一个可行解。

通过 `count_onward_moves()` 计算落点的可走步数:

```
static int count_onward_moves(const Board& board, int x, int y) {
    int count = 0;
    for (int i = 0; i < 8; ++i) {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (nx >= 0 && nx < BOARD_SIZE && ny >= 0 && ny < BOARD_SIZE && board(nx, ny) == 0) {
            ++count;
        }
    }
    return count;
}
```

1.1 算法

在 solve_heuristic(Point start) 内部对每一步的 MoveOption 数组排序，并向最小的方向进发。

```
// 枚举所有下一步的可选走法
for (int i = 0; i < 8; ++i) {
    int nx = x + dx[i];
    int ny = y + dy[i];
    if (nx >= 0 && nx < BOARD_SIZE && ny >= 0 && ny < BOARD_SIZE && board(nx, ny) == 0) {
        int onward = count_onward_moves(board, nx, ny);
        options.push_back({nx, ny, onward});
    }
}
// 按后继步数升序排序
std::sort(options.begin(), options.end(), [](const MoveOption& a, const MoveOption& b) {
    return a.onward < b.onward;
});
```

但贪心算法只能得到一个可行解。我们希望算法可以找到多条可行解（具有找到全部可行解的潜力）。

1.1.3 基于 Warnsdorf's Rule 的深度搜索算法

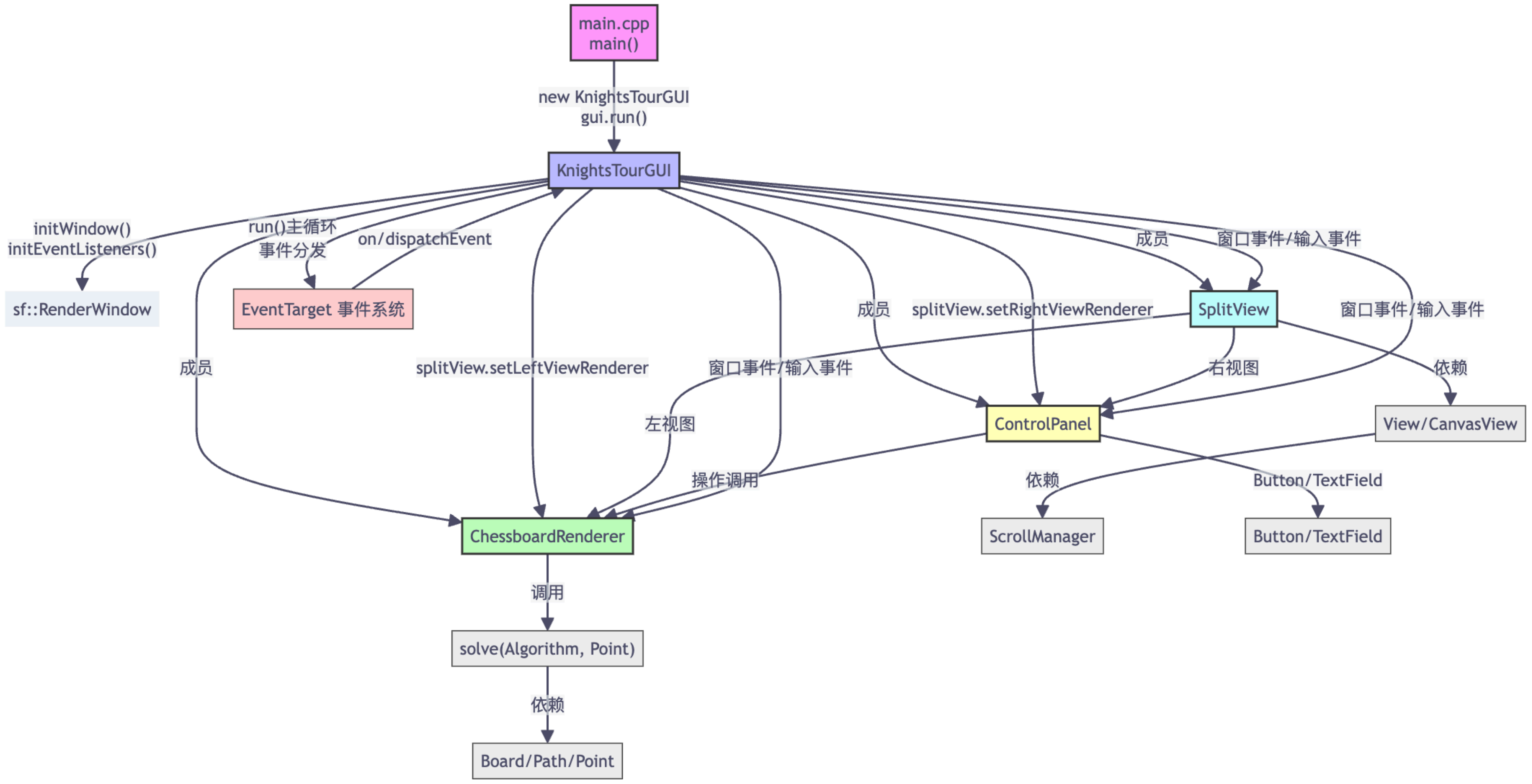
算法实现 solve_heuristic_enhancer(Point start):

首次到达某结点时，基于 Warnsdorf's Rule 对其可行方向排序，优先选择出路最少的方向从而减少回溯次数。

由于框架仍是深度优先搜索，如果想找到多条可行路径，只需在找到一条可行路径后继续回溯即可。

```
while (!stk.empty()) {
    if (step == BOARD_SIZE * BOARD_SIZE) {
        if (!stk.empty()) {...} // 保存最后一步结果
        if(countHistory == NUM_OF_PATH) break;
        else {// 后退一步继续搜其他路径
            Point end = stk.peek().pos;
            stk.pop();  --step; board(end.x, end.y) = 0;
            Point start = stk.empty() ? end : stk.peek().pos;
            history.push_back({end, start, 0});
            continue;
        }
}
```

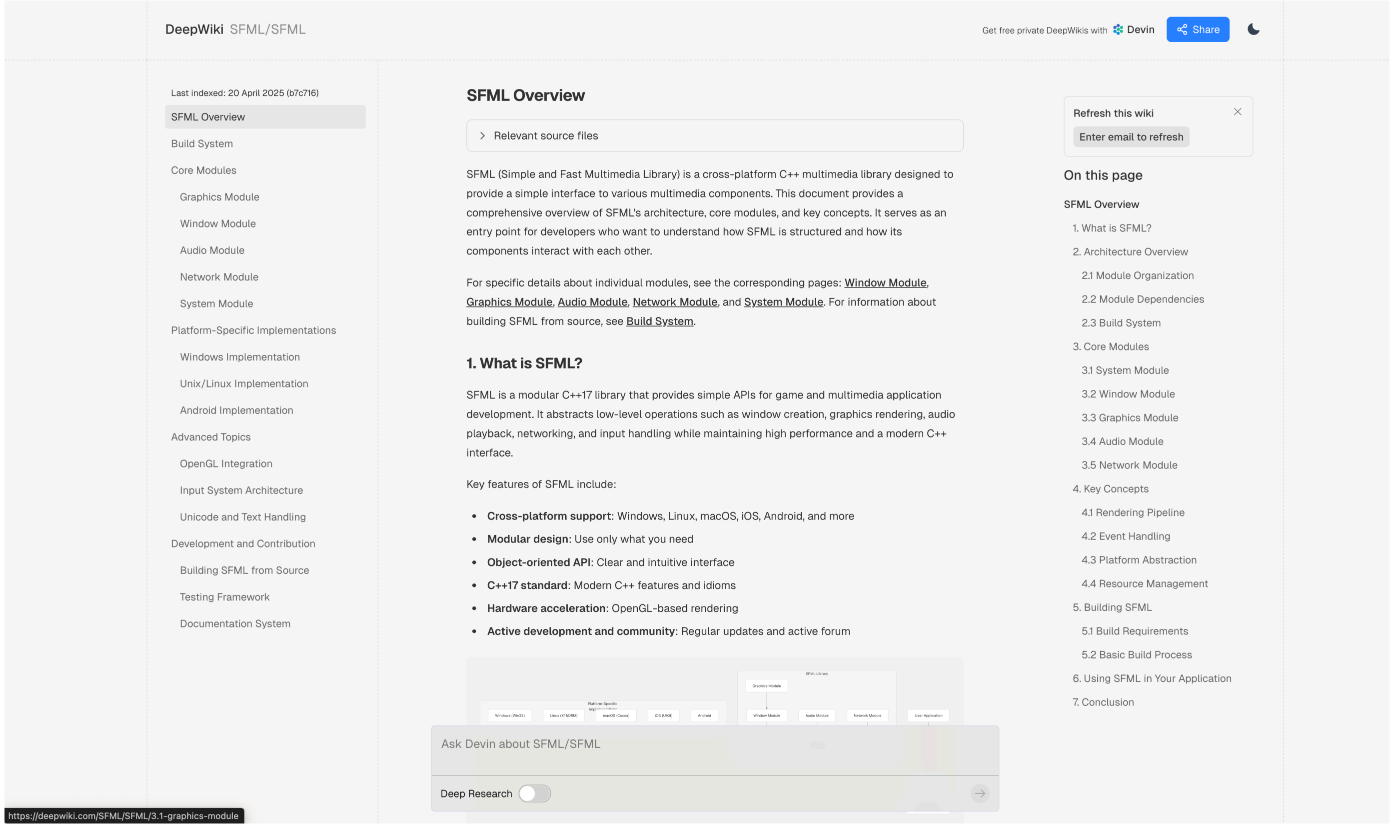
1.2 GUI



main.cpp
main()
new KnightsTourGUI
gui.run()
KnightsTourGUI
initWindow()
initEventListeners()
run()主循环
事件分发
on/dispatchEvent
成员
窗口事件/输入事件
sf::RenderWindow
EventTarget 事件系统
成员
splitView.setRightViewRenderer
SplitView
窗口事件/输入事件
成员
splitView.setLeftViewRenderer
窗口事件/输入事件
右视图
依赖
左视图
ControlPanel
View/CanvasView
操作调用
依赖
Button/TextField
ChessboardRenderer
ScrollManager
Button/TextField
调用
solve(Algorithm, Point)
依赖
Board/Path/Point

1.2 GUI

使用第三方库 SFML



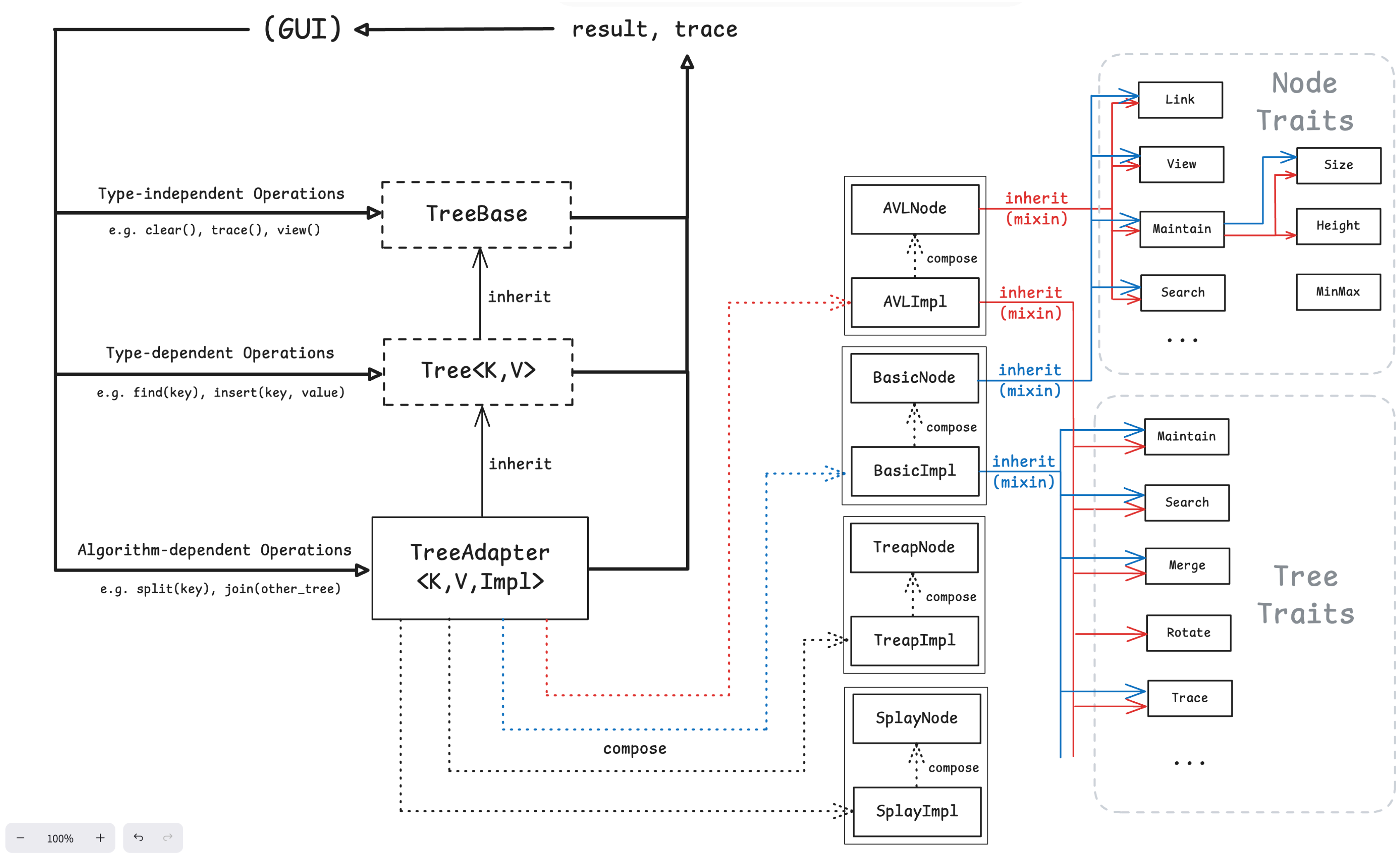
1.2 GUI

一个 bug

```
ChessboardRenderer::ChessboardRenderer(sf::Font& font)
    : font(font),
      running(false),
      ...
      currentStep(0),
      animProgress(0.0f),
      knight(knightTexture), // knightTexture not constructed!
      lastRenderedStep(-1),
      lastRenderedSolution(-1) {
```

2. 平衡树操作演示

2.1 算法



(GUI)
result, trace
Type-independent Operations
e.g. clear(), trace(), view()
TreeBase
inherit
Type-dependent Operations
e.g. find(key), insert(key, value)
Tree<K,V>
inherit
Algorithm-dependent Operations
e.g. split(key), join(other_tree)
TreeAdapter
<K,V,Impl>
compose
AVLNode
compose
AVLImpl
BasicNode
compose
BasicImpl
TreapNode
compose
TreapImpl
SplayNode
compose
SplayImpl
inherit
(mixin)
inherit
(mixin)
inherit
(mixin)
inherit
(mixin)
Node
Traits
Link
View
Maintain
Search
...
Size
Height
MinMax
Tree
Traits
Maintain
Search
Merge
Rotate
Trace
...
100%

2.1 算法

2.1.1 基础操作实现

例：实现 find（trait::node::Search, trait::Search）

```
template <typename Node> struct Search {
    auto find(auto&& key) {
        // ...
    }
};
template <typename Tree> struct Search {
    auto find(auto&& key) {
        auto&& root = static_cast<const Tree*>(this)->root;
        return root ? root->find(key) : nullptr;
    }
};
struct BasicTreeImpl : Search<BasicTreeImpl>;
struct AVLTreeImpl : Search<AVLTreeImpl>;
struct TreapImpl : Search<TreapImpl>;
```

2.1 算法

例：实现可提供不同功能的 maintain()（trait::node::Maintain）

```
template <typename Node> struct Height {
    int height{1};
    void maintain() {
        auto& self = *(static_cast<Node*>(this));
        auto l = self.child[L] ? self.child[L]->height : 0;
        auto r = self.child[R] ? self.child[R]->height : 0;
        self.height = 1 + std::max(l, r);
    }
};
template <typename Node> struct Size {
    size_t size{1};
    void maintain() {
        auto& self = *(static_cast<Node*>(this));
        auto l = self.child[L] ? self.child[L]->size : 0;
        auto r = self.child[R] ? self.child[R]->size : 0;
        self.size = 1 + l + r;
    }
};
```

2.1 算法

```cpp
// helper trait to maintain multiple properties
template <typename... Ts> struct Maintain : Ts... {
    void maintain() { (Ts::maintain(), ...); }
};

// imports a maintain() that maintains size
struct BasicNode : Maintain<Size<BasicNode>>;
// imports a maintain() that maintains both size and height
struct AVLNode : Maintain<Size<AVLNode>, Height<AVLNode>>;
```

2.1.2 旋转实现（trait::Rotate）

```cpp
template <typename Tree> struct Rotate {
    void rotate(int dir, auto& root) {
        auto& self = *static_cast<Tree*>(this);
        auto new_root = self.unbind(root, dir ^ 1);
        if (new_root->child[dir]) {
            self.bind(root, dir ^ 1, self.unbind(new_root, dir));
        }
        auto parent = root->parent;
        self.bind(new_root, dir, std::move(root));
        self.moveNode(root, std::move(new_root), parent);
        root->child[dir]->maintain();
        root->maintain();
    }
    void rotateL(auto& root) { return rotate(L, root); }
    void rotateR(auto& root) { return rotate(R, root); }
    void rotateLR(auto& root) {
        rotateL(root->child[L]), rotateR(root);
    }
    void rotateRL(auto& root) {
        rotateR(root->child[R]), rotateL(root);
    }
};
```

2.1 算法

2.1.3 AVL 树的 `split` 和 `join` (`AVLTree::{join, split}`)

- 先实现 `join(left_tree, sperator_node, right_tree)`：给定 key 值不交的两棵 AVL 树和一个 key 值在两树之间的分界点节点，合并成一棵树

  考虑 $\text{height}_{\text{left}} \geq \text{height}_{\text{right}}$ 的情况，反之对称

  在左树中找到高度为 h_{right} 或 $h_{\text{right}} + 1$ 的点 `cut_tree`，由于左树是 AVL 树，一定能找到

  将 `cut_tree` 和 `right_tree` 挂到 `seperator_node` 上，然后放回原先的位置

  高度最多改变 1，从 `cut_tree` 位置向上维护平衡即可。

  时间复杂度：$O\left(\left|h_{\text{left}} - h_{\text{right}}\right|\right)$

2.1 算法

2.1.3 AVL 树的 `split` 和 `join` (`AVLTree::{join, split}`)

- 先实现 `join(left_tree, sperator_node, right_tree)`：给定 key 值不交的两棵 AVL 树和一个 key 值在两树之间的分界点节点，合并成一棵树

  考虑 $\text{height}_{\text{left}} \geq \text{height}_{\text{right}}$ 的情况，反之对称

  在左树中找到高度为 h_{right} 或 $h_{\text{right}} + 1$ 的点 `cut_tree`，由于左树是 AVL 树，一定能找到

  将 `cut_tree` 和 `right_tree` 挂到 `seperator_node` 上，然后放回原先的位置

  高度最多改变 `1`，从 `cut_tree` 位置向上维护平衡即可。

  时间复杂度：$O\left(\left|h_{\text{left}} - h_{\text{right}}\right|\right)$

- `join(left_tree, right_tree)`:

  删除 `left_tree.max()` 或 `right_tree.min()`，转换为带 seperator 的 `join`

2.1 算法

- `split(tree, key)`

  如右图所示，在 `find(key)` 的路径上的位置将节点和它的左右子树分开

  然后自底向上合并，每一次合并用路径中的点（图中的 P_i）作为 seperator 合并两子树

$$\alpha P \longleftarrow \mathrm{join}(\alpha, {\color{red}P})$$

$$\beta P_8 \beta_8 \longleftarrow \mathrm{join}(\beta, {\color{red}P_8}, \beta_8)$$

$$\alpha_7 P_7 \alpha P \longleftarrow \mathrm{join}(\alpha_7, {\color{red}P_7}, \alpha P)$$

$$\ldots \longleftarrow \ldots$$

  每一次合并的复杂度是高度差，高度差之和不超过总高度，所以复杂度为 $O(\log n)$

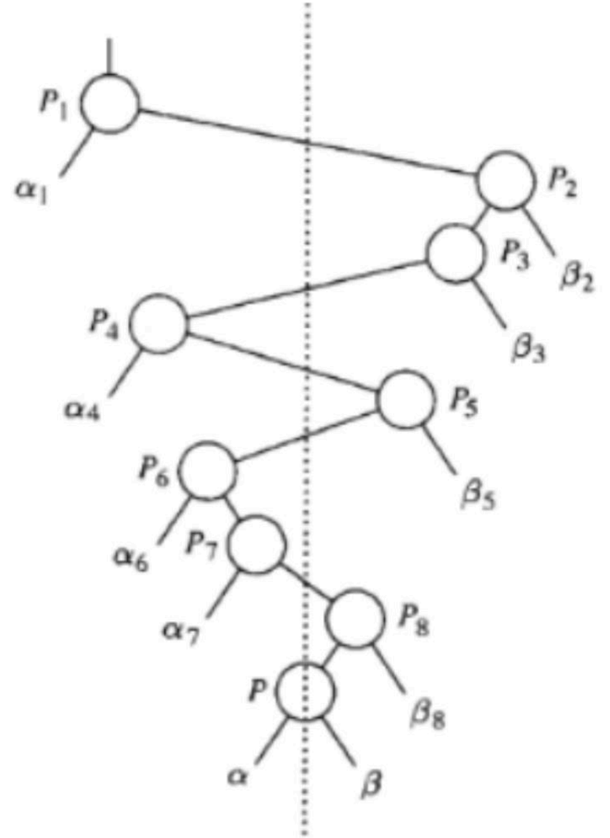


图 4　split 演示. ref.TAOCP

2.2 实现细节

```
struct TreeBase {
    virtual ~TreeBase() = default;
    virtual auto size() const -> size_t = 0;
    virtual void clear() = 0;
    virtual auto view() const -> ForestView = 0;
    virtual auto trace() -> std::vector<ForestView> = 0;
    virtual auto trace(const std::function<void()>& func) -> std::vector<ForestView> = 0;
    virtual void traceStart() = 0;
    virtual void traceStop() = 0;
    virtual void printCLI() const = 0;
    virtual auto stringify() const -> std::string = 0;
    virtual auto name() const -> std::string = 0;
};
```

2.2 实现细节

```cpp
template <typename K, typename V> struct Tree : TreeBase {
    virtual auto find(const K& key) -> Pair<const K, V>* = 0;
    virtual auto findKth(size_t rank) -> Pair<const K, V>* = 0;
    virtual auto min() -> Pair<const K, V>* = 0;
    virtual auto max() -> Pair<const K, V>* = 0;
    virtual auto insert(const K& key, const V& value) -> Status = 0;
    virtual auto remove(const K& key) -> Status = 0;
    virtual void traverse(const std::function<void(const K&, V&)>& func) = 0;
    virtual auto operator[](const K& key) -> V& = 0;
    virtual auto operator[](const K& key) const -> const V& = 0;
};

template <typename K, typename V, template <typename, typename> typename Impl>
struct TreeAdapter : Tree<K, V> {
    friend struct Test;
    auto size() const -> size_t override { return impl->size(); }
    auto view() const -> ForestView override { return impl->view(); }
    ...
    std::unique_ptr<Impl<K, V>> impl;
};
template <typename K, typename V>
using AVLTree = TreeAdapter<K, V, AVLTreeImpl>;
```

2.2 实现细节

2.2.1 内存管理

使用 `std::unique_ptr` 管理节点所有权，防止内存泄露或者 `double free` 问题

2.2 实现细节

2.2.1 内存管理

使用 `std::unique_ptr` 管理节点所有权，防止内存泄露或者 `double free` 问题

2.2.2 Trace 记录 (`trait::Trace`)

- 在结构体中放一个 `std::vector<ForestView> record;` 记录每一步操作之后的状态

  所有对树结构的操作都通过调用 `bind()`, `unbind()` 方法，内部自动维护以及记录 trace

2.2 实现细节

2.2.1 内存管理

使用 `std::unique_ptr` 管理节点所有权，防止内存泄露或者 `double free` 问题

2.2.2 Trace 记录 (`trait::Trace`)

- 在结构体中放一个 `std::vector<ForestView> record;` 记录每一步操作之后的状态

  所有对树结构的操作都通过调用 `bind()`, `unbind()` 方法，内部自动维护以及记录 trace

- 记录 trace 的方法：

  维护当前森林的根节点列表 `std::set<Node*> entries;`

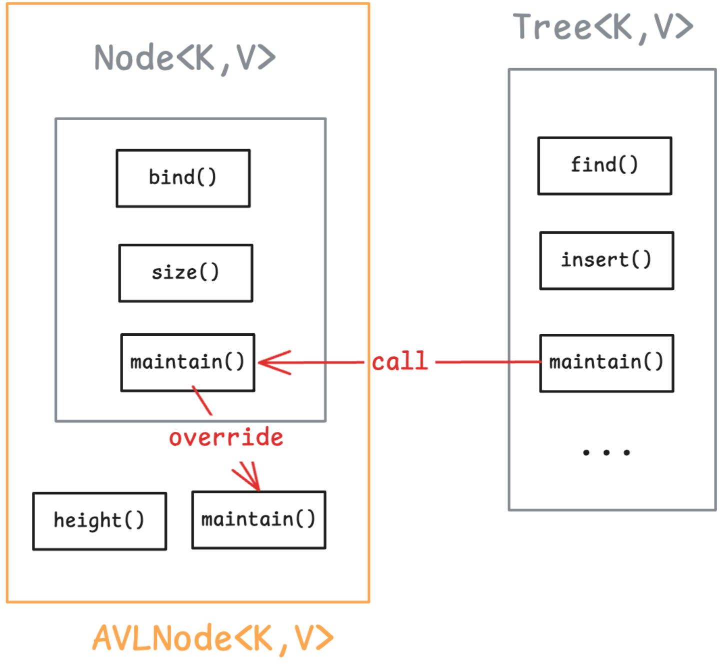
  每作一次记录 `snapshot()` 就复制出 `entries` 对应每一颗树中的信息，保存至 `record`

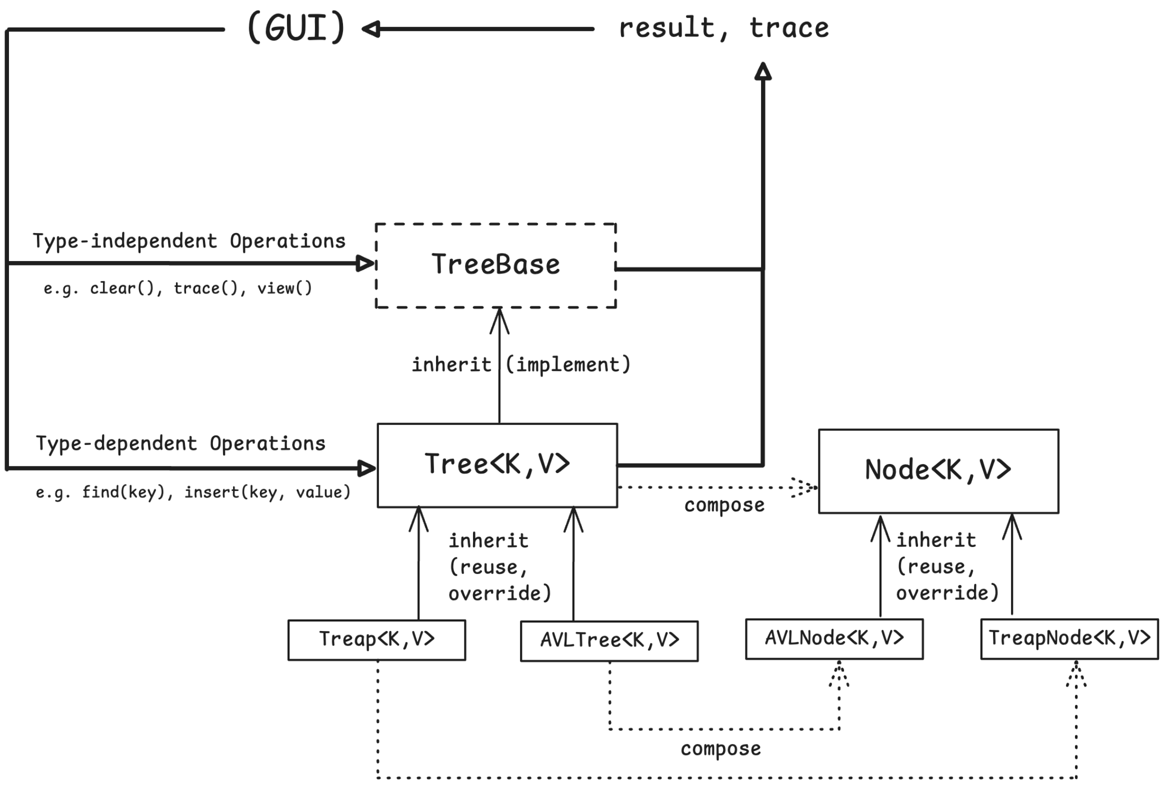
2.2 实现细节

2.2.3 性能优化

最开始的结构：

- 非常容易想到





2.2 实现细节

但

- 以 `maintain()`为例，每一次自底向上维护信息时，都需要调用 `node::maintain()`，然而这是一个虚函数，但没法内联，每一次调用都有额外开销

2.2 实现细节

但

- 以 `maintain()` 为例，每一次自底向上维护信息时，都需要调用 `node::maintain()`，然而这是一个虚函数，但没法内联，每一次调用都有额外开销
- Tree 中只会存一个基类的 `Node` 指针，每一次使用子类 `Node` 特有信息时都需要 `static_cast` 或者 `dynamic_cast`

2.2 实现细节

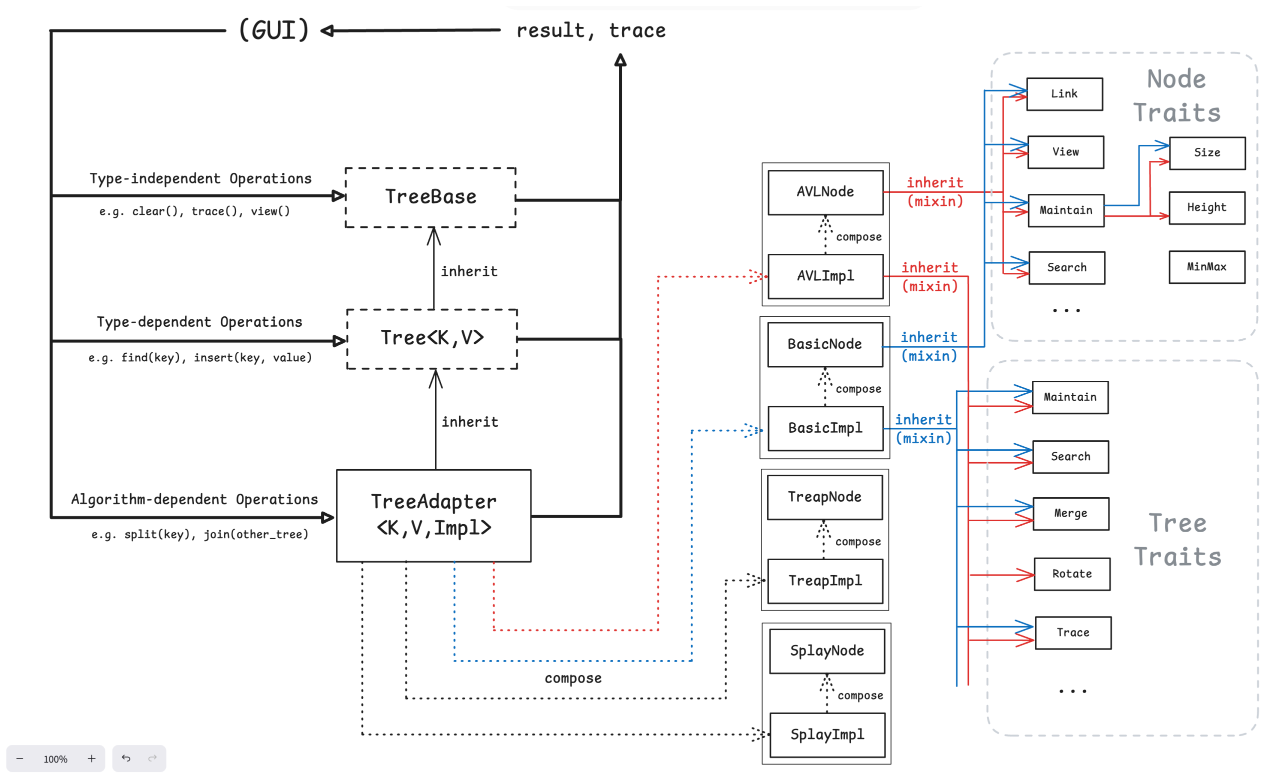
但

- 以 `maintain()` 为例，每一次自底向上维护信息时，都需要调用 `node::maintain()`，然而这是一个虚函数，但没法内联，每一次调用都有额外开销
- Tree 中只会存一个基类的 `Node` 指针，每一次使用子类 `Node` 特有信息时都需要 `static_cast` 或者 `dynamic_cast`
- 拓展功能比较麻烦
  - `Splay` 和 `AVL` 都可以旋转，要么实现两遍，要么创建一个 `RotatableTree`，增加继承层级
  - 更好的想法应该是把 `Rotate` 抽出来作为一个只提供旋转功能的 trait
  - 既然如此，为什么不把所有的功能都抽出来？

2.2 实现细节

重构!

- 所有树平级，
  复用的功能只由 trait 提供
- 由于不同的树之间没有子类型关系，需要一个 TreeAdapter 来绑定到相同的接口上



2.2 实现细节

查看是否内联：

- 重构前

```
$ nm | c++filt
000000010001e7c8 legacy::AVLTree<int, int>::AVLNode::maintain()
000000010001c4ec legacy::Tree<int, int>::Node::maintain()
```

只有 `Tree::maintain()` 被内联了，Node 本身的 `maintain()` 没有被内联

- 重构后：

```
$ nm | c++filt
0000000100020578 AVLNode<int, int>::stringify() const
000000010001fe4c AVLNode<int, int>::~AVLNode()
```

```
Tree                       Insert       Find       Remove
legacy::AVLTree(ms)         48.40      11.65        49.86
AVLTree(ms)                 32.35      10.21        41.70
std::map(ms)                25.09      11.58        30.74

CRTP Improvement(%)         33.15      12.32        16.35
```

2.2 实现细节

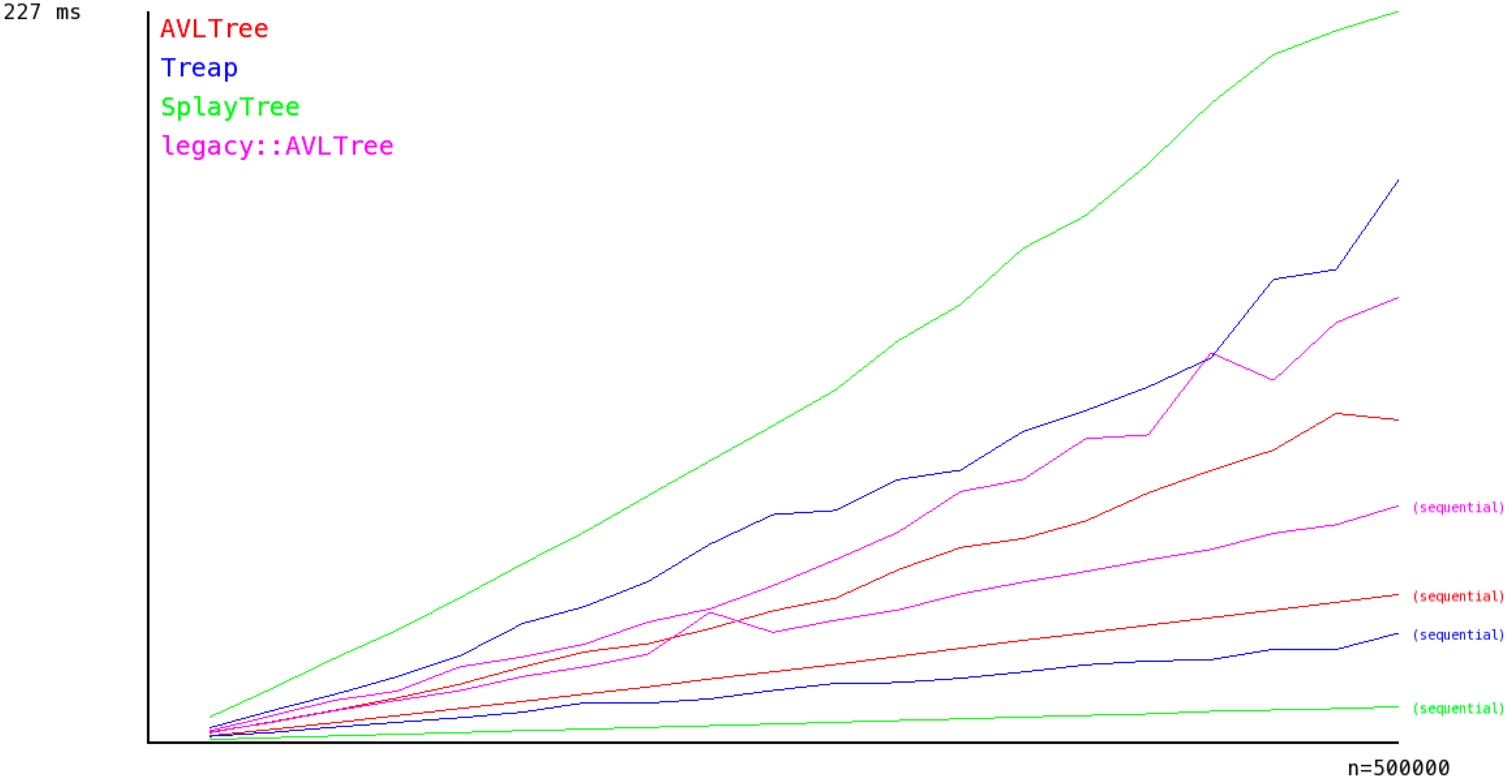


图 8 benchmark: insert

Thanks!